Matching Logic — Extended Abstract*

Grigore Roşu

University of Illinois at Urbana-Champaign, USA
`grosu@illinois.edu`

Abstract

This paper presents *matching logic*, a first-order logic (FOL) variant for specifying and reasoning about structure by means of patterns and pattern matching. Its sentences, the *patterns*, are constructed using *variables*, *symbols*, *connectives* and *quantifiers*, but no difference is made between function and predicate symbols. In models, a pattern evaluates into a power-set domain (the set of values that *match* it), in contrast to FOL where functions and predicates map into a regular domain. Matching logic uniformly generalizes several logical frameworks important for program analysis, such as: propositional logic, algebraic specification, FOL with equality, and separation logic. Patterns can specify separation requirements at any level in any program configuration, not only in the heaps or stores, without any special logical constructs for that: the very nature of pattern matching is that if two structures are matched as part of a pattern, then they can only be spatially separated. Like FOL, matching logic can also be translated into pure predicate logic, at the same time admitting its own sound and complete proof system. A practical aspect of matching logic is that FOL reasoning remains sound, so off-the-shelf provers and SMT solvers can be used for matching logic reasoning. Matching logic is particularly well-suited for reasoning about programs in programming languages that have a rewrite-based operational semantics.

1998 ACM Subject Classification D.2.4 Software/Program Verification; D.3.1 Formal Definitions and Theory; F.3 LOGICS AND MEANINGS OF PROGRAMS; F.4 MATHEMATICAL LOGIC AND FORMAL LANGUAGES

Keywords and phrases Program logic; First-order logic; Rewriting; Verification

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.x

1 Introduction, Motivation and Overview

In their simplest form, as term templates with variables, patterns abound in mathematics and computer science. They match a concrete, or ground, term if and only if there is some substitution applied to the pattern's variables that makes it equal to the concrete term, possibly via domain reasoning. This means, intuitively, that the concrete term obeys the structure specified by the pattern. We show that when combined with logical connectives and variable constraints and quantifiers, patterns provide a powerful means to specify and reason about the structure of states, or configurations, of a programming language.

Matching logic was inspired from the domain of programming language semantics, specifically from attempting to use rewrite-based operational semantics for program verification. For example, a series of large and complete semantic definitions of real languages has been recently developed using the $\mathbb{K}$ framework (`http://kframework.org` [18, 19]), such as C11 (POPL'12 [6], PLDI'15 [8]), Java 1.4 (POPL'15 [2]), JavaScript ES5 (PLDI'15 [13]), with

* The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014-2015, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

```
struct listNode { int val; struct listNode *next; };

void list_read_write(int n) {
rule  $\langle \$ \Rightarrow \texttt{return;}\ \cdots\rangle_{\mathsf{k}}\ \langle \mathsf{A} \Rightarrow \cdot\ \cdots\rangle_{\mathsf{in}}\ \langle\cdots\ \cdot \Rightarrow \mathsf{rev}(\mathsf{A})\rangle_{\mathsf{out}}$  requires  $\texttt{n} = \mathsf{len}(\mathsf{A})$
  int i=0;
  struct listNode *x=0;
inv  $\langle \beta\ \cdots\rangle_{\mathsf{in}}\ \langle\cdots\ \mathsf{list}(\texttt{x},\alpha)\ \cdots\rangle_{\mathsf{heap}} \wedge \texttt{i} \leq \texttt{n} \wedge \mathsf{len}(\beta) = \texttt{n} - \texttt{i} \wedge \mathsf{A} = \mathsf{rev}(\alpha)@\beta$
  while (i < n) {
    struct listNode *y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1; }
inv  $\langle\cdots\ \alpha\rangle_{\mathsf{out}}\ \langle\cdots\ \mathsf{list}(\texttt{x},\beta)\ \cdots\rangle_{\mathsf{heap}} \wedge \mathsf{rev}(\mathsf{A}) = \alpha@\beta$
  while (x) {
    struct listNode *y;
    y = x->next;
    printf("%d␣",x->val);
    free(x);
    x = y; }
}
```

Figure 1 Reading, storing, and reverse writing a sequence of integers

many other similar but partial semantics of other languages. Each of these language semantics has more than 1,000 semantic rules and has been thoroughly tested on benchmarks and test suites that implementations of these languages use to test their conformance, where available. Unfortunately, the current state-of-the-art in program verification is to define yet another semantics for these languages, amenable for reasoning about programs, such as an axiomatic or a dynamic logic semantics, because the general belief is that operational semantics are too low level for program verification. Moreover, when the correctness of the verifier itself is a concern, tedious proofs of equivalence between the operational and the alternative semantics are produced. That is because operational semantics are comparatively much easier to define and at the same time are executable (and thus also testable), so they are often considered as reference models of the corresponding languages, while the alternative semantics devised for verification purposes tend to be more mathematically involved and are not executable so they may hide tricky errors. Defining even one semantics for a real language like C or Java is already a huge effort. Defining more semantics, each good for a different purpose, is at best very uneconomical, with or without proofs of equivalence with the reference semantics.

Matching logic was born from our firm belief that programming languages must have formal definitions of their syntax and semantics, and that all the execution and analysis tools for a given language, such as parsers, interpreters, compilers, state-space explorers, model checkers, deductive program verifiers, etc., can be derived from just *one* reference formal definition of the language, which is executable and easy to test. No other semantics for the same language should be needed. This is the ideal scenario and we believe that there is enough evidence that it is within our reach in the short term. The main idea is that semantic rules match and apply on program *configurations*, which are algebraic data types defined as terms constrained by equations capturing the needed mathematical domains, such as lists (e.g., for input/output buffers, function stacks, etc.), sets (e.g., for concurrent threads or processes, for resources held, etc.), maps (e.g., for environments, heaps, etc), and so on.

To reason about programs we need to be able to reason about program configurations. Specifically, we need to define configuration abstractions and reason with them. Consider, for example, the program in Figure 1 which shows a C function that reads `n` elements from the standard input and prints them to the standard output in reversed order (for now, we can ignore the specifications, which are grayed). While doing so, it allocates a singly linked list storing the elements as they are read, and then deallocates the list as the elements are printed. In the end, the heap stays unchanged. To state the specification of this program, we need to match an abstract sequence of `n` elements in the input buffer, and then to match its reverse at the end of the output buffer when the function terminates. Further, to state the invariants of the two loops we need to identify a singly linked pattern in the heap, which is a partial map. Many such sequence or map patterns, as well as operations on them, can be easily defined using conventional algebraic data types (ADTs). But some of them cannot.

A major limitation of ADTs and of FOL is that operation symbols are interpreted as functions in models, which sometimes is insufficient. E.g., a two-element linked list in the heap (we regard heaps as maps from locations to values) starting with location 7 and holding values 9 and 5, written as pattern $\mathit{list}(7, 9 \cdot 5)$, can allow infinitely many heap values, one for each location where the value 5 may be stored. So we cannot define *list* as an operation symbol $\mathit{Int} \times \mathit{Seq} \to \mathit{Map}$. The FOL alternative is to define *list* as a predicate $\mathit{Int} \times \mathit{Seq} \times \mathit{Map}$, but mentioning the map all the time as an argument makes specifications verbose and hard to read, use and reason about. An alternative offered by separation logic [11, 14, 12] is to fix and move the map domain from explicit in models to implicit in the core of the logic, so that $\mathit{list}(7, 9 \cdot 5)$ is interpreted as a predicate but the map and the non-deterministic choices are implicit in the logic. We then may need custom separation logics for different languages that require different variations of map models or different configurations making use of different kinds of resources. This may also require specialized separation logic theorem provers needed for each, or otherwise encodings that need to be proved correct. Matching logic avoids the limitations of both approaches above, by interpreting its terms/formulae as *sets* of values.

Matching logic's formulae, or *patterns*, are defined using variables, symbols from a signature, and FOL connectives and quantifiers. We only treat the many-sorted first-order case here, but the same ideas can be extended to order-sorted or higher-order contexts. Specifically, if (S, Σ) is a many-sorted signature and Var an S-sorted set of variables, then a pattern φ of sort $s \in S$ can inductively be a variable in Var_s, or have the form $\sigma(\varphi_1, \ldots, \varphi_n)$ where $\sigma \in \Sigma$ is a symbol of result s (and arguments of any sorts) and $\varphi_1, \ldots, \varphi_n$ are patterns of appropriate sorts, or $\neg\varphi'$ or $\varphi' \wedge \varphi''$ or $\exists x.\varphi'$ where φ' and φ'' are patterns of sort s and $x \in \mathit{Var}$ (of any sort). Derived constructs $\vee$, $\forall$, $\to$, $\leftrightarrow$, $\top$, $\bot$ can be defined as usual. One way to think of patterns is that they collapse the function and predicate symbols of FOL, allowing patterns to be simultaneously regarded *both* as terms and as predicates. When regarded as terms they build structure, and when regarded as predicates they express constraints.

Semantically, a *model* M consists of a carrier M_s for each sort s, like in FOL, but interprets symbols $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ as maps $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ yielding a *set* of elements. In particular, σ_M can be a function, when the set contains only one element, or a partial function, when the set contains at most one element. Any *M-valuation* $\rho : \mathit{Var} \to M$ extends to a map $\overline{\rho}$ taking patterns to sets of values, where $\neg$ is interpreted as the complement, $\wedge$ as intersection, and $\exists$ as union over all compatible valuations. If φ is a pattern and $a \in \overline{\rho}(\varphi)$ then we say that *a matches φ (with ρ)*. The name of matching logic was inspired from the case when M is a term model, quite common in the context of programming language semantics where M typically represents a program configuration or a fragment of it. In that case, if a is a ground term and φ is a term with variables, then "a matches φ" in matching logic becomes precisely the usual notion of pattern matching. Pattern φ is *valid in M* iff $\overline{\rho}(\varphi) = M$ (i.e., it is matched by all elements of M), and it is *valid* iff it is valid in all models.

It turns out that, unlike in FOL, equality can be defined in matching logic (Section 4.3):

i.e., $\varphi_1 = \varphi_2$ is a pattern so that, given any model M and any M-valuation $\rho : \mathit{Var} \to M$, $\varphi_1 = \varphi_2$ is either matched by all elements when $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$, or by none otherwise.

Let us discuss some simple examples. If Σ is the signature of Peano natural numbers and M is the model of natural numbers with 0 and *succ* interpreted accordingly, then the pattern $\exists x\,.\,\mathit{succ}(x)$ is matched by all positive numbers: indeed, by the semantics of the existential quantifier, it is the *union* of all successors of natural numbers. If we want to only allow models in which 0 is interpreted as one element, *succ* as a total and injective function, and whose elements are either zero or successors of other elements, then we add the axioms:

$$\begin{array}{ll} \exists y\,.\,0 = y & 0 \vee \exists x\,.\,\mathit{succ}(x) \\ \exists y\,.\,\mathit{succ}(x) = y & \mathit{succ}(x_1) = \mathit{succ}(x_2) \to x_1 = x_2 \end{array}$$

We can go further and axiomatize *plus* the same way we are used to in algebraic specification:

$$\mathit{plus}(0, y) = y \qquad \mathit{plus}(\mathit{succ}(x), y) = \mathit{succ}(\mathit{plus}(x, y))$$

or equivalently as the following equality matching logic (and not FOL) pattern:

$$\mathit{plus}(x, y) = (x = 0 \wedge y \vee \exists z\,.\,x = \mathit{succ}(z) \wedge \mathit{succ}(\mathit{plus}(z, y)))$$

We next define a matching logic specification whose symbols are not all functions anymore. Consider a typical ADT of maps from natural to integer numbers, with *emp* the empty map, $\_ \mapsto \_$ a one binding map, $\_ \mapsto [\_]$ a map of consecutive bindings, and $\_ * \_$ the partial function merging two maps (notations inspired from separation logic [11, 14, 12]). In addition to the usual unit, associativity and commutativity axioms for *emp* and $\_ * \_$, we also add

$$\neg(0 \mapsto a) \qquad x \mapsto a * x \mapsto b = \bot \qquad x \mapsto [\epsilon] = \mathit{emp} \qquad x \mapsto [a, S] = x \mapsto a * (x + 1) \mapsto [S]$$

The first pattern says 0 cannot serve as the key of any binding. The second pattern says that the keys of different bindings in a map must be distinct. The last two patterns desugar the consecutive binding construct. Consider now a symbol $\mathit{list} \in \Sigma_{\mathit{Nat} \times \mathit{Seq}, \mathit{Map}}$ taking a number x and a sequence of integers S to a set of maps $\mathit{list}(x, S)$, together with the following:

$$\mathit{list}(0, \epsilon) = \mathit{emp} \qquad \mathit{list}(x, n \cdot S) = \exists z\,.\,x \mapsto [n, z] * \mathit{list}(z, S)$$

This looks similar to how the list predicate is defined in separation logic using recursive predicates, although in matching logic there are no predicates and no recursion. The equations above use the same principle to define *list* as the Peano equations did to define *plus*: pattern equations. We can now show (see Section 4.7) that in the model whose *Map* carrier consists of the finite-domain partial maps, and where $\mapsto$ and $*$ are interpreted appropriately, the interpretation of $\mathit{list}(x, S)$ is precisely the set of all singly-linked lists starting with $x \neq 0$ and comprising the sequence of integers S. That is, in the intended model, the $\mathit{list}(x, S)$ pattern is matched by precisely the desired lists. In fact, we can show that the matching logic specification above, in the map model, is equivalent to separation logic (Section 4.9).

Using the generic proof system of matching logic in Section 5, we can now derive properties about lists in this specification, such as, e.g., $(1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1) \to \mathit{list}(7, 9 \cdot 5)$:

$$\begin{array}{lcll} 1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 & = & 1 \mapsto [5, 0] * 7 \mapsto [9, 1] & = \\ 1 \mapsto [5, 0] * \mathit{list}(0, \epsilon) * 7 \mapsto [9, 1] & \to & (\exists z\,.\,1 \mapsto [5, z] * \mathit{list}(z, \epsilon)) * 7 \mapsto [9, 1] & = \\ \mathit{list}(1, 5 \cdot \epsilon) * 7 \mapsto [9, 1] & = & \mathit{list}(1, 5) * 7 \mapsto [9, 1] & \to \\ \exists z\,.\,7 \mapsto [9, z] \wedge \mathit{list}(z, 5) & = & \mathit{list}(7, 9 \cdot 5) & \end{array}$$

The benefits of matching logic can be perhaps best seen when there are no immediate existing logics to reason about certain structures. Consider, e.g., the operational semantics of a real language like C, whose configuration can be defined with ordinary ADTs but has more than 100 semantic cells [6, 8]. The semantic cells, written using symbols $\langle ... \rangle_{\mathsf{cell}}$, can be nested and their grouping is associative and commutative. There is a top cell $\langle ... \rangle_{\mathsf{cfg}}$ holding a subcell $\langle ... \rangle_{\mathsf{heap}}$ among many others. We can globalize the local reasoning above to the entire C configuration proving the following property using the same proof system in Section 5:

$$\forall c\!: Cfg. \forall h\!: Map\,.\, (\langle\langle 1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}} \ \rightarrow\ \langle\langle \mathit{list}(7, 9 \cdot 5) * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}})$$

Quantification over the heap or over the configuration are first-order in matching logic. We refer to such variables like h and c matching the remaining contents of a cell "cell" as *(structural) "cell" frames*; e.g., h is the *(structural) heap frame* and c is the *(structural) configuration frame*, and write them as ellipses ("...") when their particular name is irrelevant.

The C semantics consists of more than 2,000 rewrite rules between patterns (ordinary terms with variables are patterns). We are currently developing an extension of the $\mathbb{K}$ framework that allows us to verify programs using a rewrite-based operational semantics of the programming language, like in [4, 16, 20]. Matching logic reasoning is used in-between semantic rewrite rule applications to re-arrange the configuration so that semantic rules match or assertions can be proved. This work-in-progress extension of $\mathbb{K}$ will be reported elsewhere. In the remainder of this section we only want to emphasize, by means of example, that in spite of its generality, matching logic can also be implemented efficiently.

Figure 1 showed a C function whose correctness can be automatically verified by our current prototype prover. The reader can check it, as well as dozens of other programs, using the online MatchC interface at `http://matching-logic.org`; this function is under the `io` folder and it takes about 150ms to verify. The rule specification of the function states its semantics/summary: the body ($) returns in the code cell $\langle\rangle_{\mathsf{k}}$ possibly followed by other code (as mentioned, "..." are structural frames, that is, universally quantified "anonymous" variables), the sequence A of size n is consumed from the prefix of the input buffer (A is rewritten to "$\cdot$", the unit of collections, possibly followed by more input), and the reversed sequence $\mathsf{rev}(\mathsf{A})$ is put at the end of the output buffer.

The first loop invariant says the pattern $\mathsf{list}(\mathsf{x}, \alpha)$ is matched somewhere in the heap, and that the sequence β of size $\mathsf{n} - \mathsf{i}$ is available in the input buffer such that A is the reverse of the sequence that x points to, $\mathsf{rev}(\alpha)$, concatenated with β. By convention, Boolean patterns like $\mathsf{i} \leq \mathsf{n}$ can be used in any context and they are either matched by all elements when they hold, or by no elements when they do not hold (Section 4.5). The variables starting with a ? are assumed existentially quantified. The invariant of the second loop says that a sequence α can be matched as a suffix of the output buffer and sequence β can be matched within a list that x points to in the heap, such that $\alpha@\beta$ is the reverse of the original sequence A. The verification of this function consists of executing the rewrite semantics of C symbolically on all paths and, each time a pattern is encountered, a pattern implication proof task is deferred to the matching logic prover. For example, the last proof task is:

$$\begin{array}{ll} & \langle\langle I\rangle_{\mathsf{in}}\ \langle O, \alpha\rangle_{\mathsf{out}}\ \langle \mathsf{list}(\mathsf{x}, \beta) * H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{cfg}} \quad \wedge \quad \mathsf{rev}(\mathsf{A}) = \alpha@\beta \quad \wedge \quad \mathsf{x} = 0 \\ \rightarrow & \langle\langle I\rangle_{\mathsf{in}}\ \langle O, \mathsf{rev}(\mathsf{A})\rangle_{\mathsf{out}}\ \langle H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{cfg}} \end{array}$$

which can be proved using the proof system in Section 5 and the given pattern axioms.

Section 2 introduces the syntax and semantics of matching logic. Section 3 shows that, like FOL with equality, matching logic also translates to predicate logic. Section 4 enumerates a series of examples, most notably showing that equality is definable. Section 5 introduces a sound and complete proof system. Section 6 discusses related work and Section 7 concludes.

2 Matching Logic

We assume the reader familiar with many-sorted sets, functions, and FOL. For any given set of sorts S, we assume Var is an S-sorted set of variables, sortwise infinite and disjoint. We may write $x : s$ instead of $x \in \mathit{Var}_s$; when the sort of x is irrelevant, we just write $x \in \mathit{Var}$. We let $\mathcal{P}(M)$ denote the powerset of a many-sorted set M, which is itself many-sorted.

▶ **Definition 1.** Let (S, Σ) be a many-sorted signature of *symbols*. Matching logic (S, Σ)-*formulae*, also called (S, Σ)-*patterns*, or just (matching logic) *formulae* or *patterns* when (S, Σ) is understood from context, are inductively defined as follows for all sorts $s \in S$:

$$\varphi_s \;::=\; x \in \mathit{Var}_s \;\mid\; \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \;\mid\; \neg\varphi_s \;\mid\; \varphi_s \wedge \varphi_s \;\mid\; \exists x . \varphi_s \text{ with } x \in \mathit{Var}$$

Let Pattern be the S-sorted set of patterns. By abuse of language, we refer to the symbols in Σ also as patterns: think of $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ as the pattern $\sigma(x_1 : s_1, \ldots, x_n : s_n)$.

To compact notation, $\varphi \in$ Pattern means φ is any pattern, while $\varphi_s \in$ Pattern or $\varphi \in$ Pattern$_s$ that it has sort s. We adopt the following derived constructs:

$$\begin{array}{rcl@{\qquad}rcl}
\bot_s & \equiv & x : s \wedge \neg x : s & \varphi_1 \rightarrow \varphi_2 & \equiv & \neg\varphi_1 \vee \varphi_2 \\
\top_s & \equiv & \neg\bot_s & \varphi_1 \leftrightarrow \varphi_2 & \equiv & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\
\varphi_1 \vee \varphi_2 & \equiv & \neg(\neg\varphi_1 \wedge \neg\varphi_2) & \forall x . \varphi & \equiv & \neg(\exists x . \neg\varphi)
\end{array}$$

and let $FV(\varphi)$ denote the *free variables* of φ, defined as usual.

▶ **Definition 2.** A *matching logic* (S, Σ)-*model* M, or simply a *model* when (S, Σ) is understood, consists of: (1) An S-sorted set $\{M_s\}_{s \in S}$, where each set M_s, called the *carrier of sort* s *of* M, is assumed non-empty; and (2) A function $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ for each symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, called the *interpretation* of σ in M.

Note that usual (S, Σ)-algebras are special cases of matching logic models, where $|\sigma_M(m_1, \ldots, m_n)| = 1$ for any $m_1 \in M_{s_1}$, $\ldots$, $m_n \in M_{s_n}$. Similarly, partial (S, Σ)-algebras also fall as special case, where $|\sigma_M(m_1, \ldots, m_n)| \leq 1$, since we can capture the undefinedness of σ_M on m_1, $\ldots$, m_n with $\sigma_M(m_1, \ldots, m_n) = \emptyset$. We tacitly use the same notation σ_M for its extension $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \rightarrow \mathcal{P}(M_s)$ to argument sets, i.e., $\sigma_M(A_1, \ldots, A_n) = \bigcup\{\sigma_M(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots, a_n \in A_n\}$, where $A_1 \subseteq M_{s_1}, \ldots, A_n \subseteq M_{s_n}$.

▶ **Definition 3.** Given a model M and a map $\rho : \mathit{Var} \rightarrow M$, called an M-*valuation*, let its extension $\overline{\rho} :$ Pattern $\rightarrow \mathcal{P}(M)$ be inductively defined as follows:

- $\overline{\rho}(x) = \{\rho(x)\}$, for all $x \in \mathit{Var}_s$
- $\overline{\rho}(\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})) = \sigma_M(\overline{\rho}(\varphi_1), \ldots \overline{\rho}(\varphi_n))$
- $\overline{\rho}(\neg\varphi_s) = M_s \setminus \overline{\rho}(\varphi_s)$ ("$\setminus$" is set difference)
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\exists x . \varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : \mathit{Var} \rightarrow M,\ \rho'\restriction_{\mathit{Var} \setminus \{x\}} = \rho\restriction_{\mathit{Var} \setminus \{x\}}\}$ ("$\rho\restriction_A$" is ρ restricted to A)

The extension of ρ works as expected with the derived constructs:

- $\overline{\rho}(\bot_s) = \emptyset$ and $\overline{\rho}(\top_s) = M_s$
- $\overline{\rho}(\varphi_1 \vee \varphi_2) = \overline{\rho}(\varphi_1) \cup \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\varphi_1 \rightarrow \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ implies } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1) \setminus \overline{\rho}(\varphi_2))$
- $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ iff } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1) \,\Delta\, \overline{\rho}(\varphi_2))$
  ("Δ" is the set symmetric difference operation)
- $\overline{\rho}(\forall x . \varphi) = \bigcap\{\overline{\rho'}(\varphi) \mid \rho' : \mathit{Var} \rightarrow M,\ \rho'\restriction_{\mathit{Var} \setminus \{x\}} = \rho\restriction_{\mathit{Var} \setminus \{x\}}\}$

▶ **Definition 4.** Model M *satisfies* φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : \mathit{Var} \rightarrow M$.

▶ Proposition 5. The following properties hold:

- If $\rho_1, \rho_2 : Var \to M$, $\rho_1\restriction_{FV(\varphi)} = \rho_2\restriction_{FV(\varphi)}$ then $\overline{\rho_1}(\varphi) = \overline{\rho_2}(\varphi)$
- If $x \in Var_s$ then $M \models x$ iff $|M_s| = 1$
- If $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and $\varphi_1, \ldots, \varphi_n$ are patterns of sorts $s_1, \ldots, s_n$, respectively, then we have $M \models \sigma(\varphi_1, \ldots, \varphi_n)$ iff $\sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_n)) = M_s$ for any $\rho : Var \to M$
- $M \models \neg\varphi$ iff $\overline{\rho}(\varphi) = \emptyset$ for any $\rho : Var \to M$
- $M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$
- If $\exists x.\varphi_s$ is closed, then $M \models \exists x.\varphi_s$ iff $\bigcup\{\overline{\rho}(\varphi_s) \mid \rho : Var \to M\} = M_s$; hence, $M \models \exists x.x$
- $M \models \varphi_1 \to \varphi_2$ iff $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$
- $M \models \varphi_1 \leftrightarrow \varphi_2$ iff $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ for all $\rho : Var \to M$
- $M \models \forall x.\varphi$ iff $M \models \varphi$

Note that property "if φ closed then $M \models \neg\varphi$ iff $M \not\models \varphi$", which holds in FOL, does not hold in matching logic. Indeed, suppose φ is a constant symbol, say 0, of sort s. Then $M \models \neg 0$ is equivalent to $0_M = \emptyset$, while $M \not\models 0$ is equivalent to $0_M \neq M_s$.

▶ **Definition 6.** Pattern φ is *valid*, written $\models \varphi$, iff $M \models \varphi$ for all M. If $F \subseteq$ PATTERN then $M \models F$ iff $M \models \varphi$ for all $\varphi \in F$. F *entails* φ, written $F \models \varphi$, iff for all M, we have $M \models F$ implies $M \models \varphi$. A *matching logic specification* is a triple (S, Σ, F) with $F \subseteq$ PATTERN.

3 Reduction to Predicate Logic

It is known that FOL formulae can be translated into equivalent predicate logic formulae, by replacing each function symbol with a predicate symbol and then systematically transforming terms into formulae. We can similarly translate patterns into equivalent predicate logic formulae. Consider pure predicate logic with equality and no constants, whose satisfaction relation is $\models^{=}_{PL}$. If (S, Σ) is a matching logic signature, let (S, Π_Σ) be the predicate logic signature with $\Pi_\Sigma = \{\pi_\sigma : s_1 \times \cdots \times s_n \times s \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$. We define the translation PL of matching logic (S, Σ)-patterns into predicate logic (S, Π_Σ)-formulae inductively as follows:

$$PL(\varphi) = \forall r \,.\, PL_2(\varphi, r)$$

$$\begin{aligned}
PL_2(x, r) &= (x = r)\\
PL_2(\sigma(\varphi_1, \ldots, \varphi_n), r) &= \exists r_1 \cdots \exists r_n \,.\, PL_2(\varphi_1, r_1) \wedge \cdots \wedge PL_2(\varphi_n, r_n) \wedge \pi_\sigma(r_1, \ldots, r_n, r)\\
PL_2(\neg\varphi, r) &= \neg PL_2(\varphi, r)\\
PL_2(\varphi_1 \wedge \varphi_2, r) &= PL_2(\varphi_1, r) \wedge PL_2(\varphi_2, r)\\
PL_2(\exists x \,.\, \varphi, r) &= \exists x \,.\, PL_2(\varphi, r)
\end{aligned}$$

$$PL(\{\varphi_1, \ldots, \varphi_n\}) = \{PL(\varphi_1), \ldots, PL(\varphi_n)\}$$

Then the following result holds, like for FOL:

▶ Proposition 7. If F is a set of patterns and φ is a pattern, then $F \models \varphi$ iff $PL(F) \models^{=}_{PL} PL(\varphi)$

Proposition 7 gives a sound and complete procedure for matching logic reasoning: translate the specification (S, Σ, F) and pattern to prove φ into the predicate logic specification $(S, \Pi_\Sigma, PL(F))$ and formula $PL(\varphi)$, respectively, and then derive it using the sound and complete proof system of predicate logic. However, translating patterns to predicate logic formulae makes reasoning harder not only for humans, but also for machines, since new quantifiers are introduced. For example, $(1 \mapsto 5{*}2 \mapsto 0{*}7 \mapsto 9{*}8 \mapsto 1) \to \mathit{list}(7, 9{\cdot}5)$ discussed and proved in Section 1, translates into the formula (to keep it small, we do not translate the numbers) $\forall r \,.\, (\exists r_1 \,.\, \exists r_2 \,.\, \pi_{\mapsto}(1, 5, r_1) \wedge (\exists r_3 \,.\, \exists r_4 \,.\, \pi_{\mapsto}(2, 0, r_3) \wedge (\exists r_5 \,.\, \exists r_6 \,.\, \pi_{\mapsto}(7, 9, r_5) \wedge \pi_{\mapsto}(8, 1, r_6) \wedge \pi_*(r_5, r_6, r_4)) \wedge \pi_*(r_3, r_4, r_2)) \wedge \pi_*(r_1, r_2, r)) \to \exists r_7 \,.\, \pi.(9, 5, r_7) \wedge \pi_{\mathit{list}}(7, r_7, r)$.

What we would like is to reason directly with matching logic patterns, the same way we reason directly with terms in FOL without translating them to predicate logic.

▶ Proposition 8. The following hold for matching logic:

1. $\models \varphi$, where φ is a propositional tautology (over patterns)
2. Modus ponens: $\models \varphi_1$ and $\models \varphi_1 \to \varphi_2$ implies $\models \varphi_2$
3. $\models (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ when $x \notin FV(\varphi_1)$
4. Universal generalization: $\models \varphi$ implies $\models \forall x \,.\, \varphi$

Proposition 8 states that the proof system of pure predicate logic is actually sound for matching logic *as is*. Section 5 shows that a few additional proof rules yield a sound and complete proof system for matching logic, similarly to how Substitution $(\forall x \,.\, \varphi \to \varphi[t/x])$ together with the four proof rules of pure predicate logic brings complete deduction to FOL. But before that, we demonstrate the usefulness of matching logic by a series of examples.

4 Examples and Notations

We have already seen some simple patterns in Section 2, such as $\exists x.x$ (satisfied by all models) and $\forall x.x$ (satisfied only by models whose carrier of the sort of x contains only one element). Here we illustrate matching logic by means of a series of more complex examples.

4.1 Propositional logic

If S contains only one sort Prop, Σ is empty, and we drop the existential quantifier, then the syntax of matching logic becomes that of propositional calculus: $\varphi ::= \mathit{Var}_{\mathit{Prop}} \mid \neg\varphi \mid \varphi \wedge \varphi$.

▶ Proposition 9. For any proposition φ, the following holds: $\models_{\mathit{Prop}} \varphi$ iff $\models \varphi$.

An alternative way to capture propositional logic is to add a constant symbol to Σ for each propositional variable, and then associate a ground pattern to each proposition. Proposition 9 still holds, despite the fact that propositional constants can be interpreted as arbitrary sets. That is since $(\mathcal{P}(M), \neg_M, \cap)$ is a model of propositional logic for any set M.

4.2 Pure predicate logic

If S is a sort set and Π is a set of predicate symbols, the syntax of pure predicate logic formulae (without equality) is $\varphi ::= \pi(x_1, \ldots, x_n)$ with $\pi \in \Pi_{s_1 \ldots s_n} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi$. We can pick a new sort name, Pred, and construct a matching logic signature $(S \cup \{\mathit{Pred}\}, \Sigma)$ where $\Sigma_{s_1 \ldots s_n, \mathit{Pred}} = \Pi_{s_1 \ldots s_n}$. Then any predicate logic formula can be trivially regarded as a matching logic pattern. The following result then holds:

▶ Proposition 10. For any predicate logic formula φ, the following holds: $\models_{PL} \varphi$ iff $\models \varphi$.

4.3 Definedness, Equality, Membership

Pattern definedness, equality and membership can be defined in matching logic, without any special support or logic extensions. Let us first discuss why we cannot use $\leftrightarrow$ as equality. Indeed, since $M \models \varphi_1 \leftrightarrow \varphi_2$ iff $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ for all $\rho : \mathit{Var} \to M$, one may be tempted to do so. E.g., given a signature with one sort and one unary symbol f, one may think that pattern $\exists y \,.\, f(x) \leftrightarrow y$ defines precisely the models where f is a function. Unfortunately, that is not true. Consider model M with $M = \{1, 2\}$ and f_M the non-function $f_M(1) = \{1, 2\}$, $f_M(2) = \emptyset$. Let $\rho : \mathit{Var} \to M$; recall (Definition 3) that ρ'extension $\overline{\rho}$ to patterns interprets "$\exists$" as union and "$\leftrightarrow$" as the complement of the symmetric difference. If $\rho(x) = 1$ then

$\overline{\rho}(\exists y\,.\,f(x) \leftrightarrow y) = (M\backslash(\{1,2\}\Delta\{1\})) \cup (M\backslash(\{1,2\}\Delta\{2\})) = \{1,2\} = M$. If $\rho(x) = 2$ then $\overline{\rho}(\exists y\,.\,f(x) \leftrightarrow y) = (M\backslash(\emptyset\Delta\{1\})) \cup (M\backslash(\emptyset\Delta\{2\})) = \{1,2\} = M$. Hence, $M \models \exists y\,.\,f(x) \leftrightarrow y$.

The problem above was that the interpretation of $\varphi_1 \leftrightarrow \varphi_2$ is not equivalent to either $\top$ or $\bot$, as we are used to think in FOL. Specifically, $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$ does not suffice for $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \emptyset$ to hold. Indeed, $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\overline{\rho}(\varphi_1)\ \Delta\ \overline{\rho}(\varphi_2))$ and there is nothing to prevent, e.g., $\overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2) \neq \emptyset$, in which case $\overline{\rho}(\varphi_1)\ \Delta\ \overline{\rho}(\varphi_2) \neq M$. What we would like to have is a proper equality, $\varphi_1 = \varphi_2$, which behaves like a predicate: $\overline{\rho}(\varphi_1 = \varphi_2) = \emptyset$ when $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$, and $\overline{\rho}(\varphi_1 = \varphi_2) = M$ when $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$. Moreover, we want equalities to be used with terms of any sort, and in contexts of any sort.

The above can be achieved methodologically in matching logic, by adding to the signature a *definedness* symbol $\lceil\_\rceil_{s_1}^{s_2} \in \Sigma_{s_1,s_2}$ for any sorts s_1 and s_2, together with the pattern axiom $\lceil x\!:\!s_1\rceil_{s_1}^{s_2}$ enforcing $(\lceil\_\rceil_{s_1}^{s_2})_M(m_1) = M_{s_2}$ in all models M for all $m_1 \in M_{s_1}$, that is, for any $\rho : \mathit{Var} \to M$, $\overline{\rho}(\lceil\varphi\rceil_{s_1}^{s_2})$ is either $\emptyset$ when $\overline{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is M_{s_2} when $\overline{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined). We can now use $\_ =_{s_1}^{s_2} \_$ and $\_ \in_{s_1}^{s_2} \_$, respectively, as aliases:

$$\begin{array}{rcll} \varphi =_{s_1}^{s_2} \varphi' & \equiv & \neg\lceil\neg(\varphi \leftrightarrow \varphi')\rceil_{s_1}^{s_2} & \text{where } \varphi, \varphi' \in \textsc{Pattern}_{s_1} \\ x \in_{s_1}^{s_2} \varphi & \equiv & \lceil x \wedge \varphi\rceil_{s_1}^{s_2} & \text{where } x \in \mathit{Var}_{s_1}, \varphi \in \textsc{Pattern}_{s_1} \end{array}$$

▶ Proposition 11. With the above, the following hold:
1. $\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = \emptyset$ iff $\overline{\rho}(\varphi) \neq \overline{\rho}(\varphi')$, and $\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = M_{s_2}$ iff $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$
2. $\models \varphi =_{s_1}^{s_2} \varphi'$ iff $\models \varphi \leftrightarrow \varphi'$
3. $\overline{\rho}(x \in_{s_1}^{s_2} \varphi) = \emptyset$ iff $\rho(x) \notin \overline{\rho}(\varphi)$; and $\overline{\rho}(x \in_{s_1}^{s_2} \varphi) = M_{s_2}$ iff $\rho(x) \in \overline{\rho}(\varphi)$
4. $\models (x \in_{s_1}^{s_2} \varphi) =_{s_2}^{s_3} (x \wedge \varphi =_{s_1}^{s_2} x)$

From now on we assume equality and membership in all specifications, without mentioning the constructions above. Moreover, since s_1 and s_2 can usually be inferred from context, we write $\lceil\_\rceil$, $=$ and $\in$ instead of $\lceil\_\rceil_{s_1}^{s_2}$, $=_{s_1}^{s_2}$, and $\in_{s_1}^{s_2}$, respectively. If the sort decorations cannot be inferred from context, then we assume the stated property/axiom/rule holds for all such sorts. For example, the generic pattern axiom "$\lceil x\rceil$ where $x \in \mathit{Var}$" replaces all the axioms $\lceil x\!:\!s_1\rceil_{s_1}^{s_2}$ above for the definedness symbol, for all the sorts s_1 and s_2. Similarly, the axiom in Section 4.7 defining list patterns within maps, $\mathit{list}(x) = (x = 0 \wedge \mathit{emp} \vee \exists z\,.\,x \mapsto z * \mathit{list}(z))$, is equivalent to the explicit axioms (for all sorts s), $\mathit{list}(x) =_{\mathit{Map}}^{s} (x =_{\mathit{Nat}}^{\mathit{Map}} 0 \wedge \mathit{emp} \vee \exists z\,.\,x \mapsto z * \mathit{list}(z))$.

Proposition 8 showed that four of the proof rule/axiom schemas of FOL are already sound for matching logic. The soundness of several others are shown below, essentially stating the soundness of the matching logic proof system, except one rule, Substitution (Section 5):

▶ Proposition 12. The following hold:
1. Equality introduction: $\models \varphi = \varphi$
2. Equality elimination: $\models \varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$
3. $\models \forall x\,.\,x \in \varphi$ iff $\models \varphi$
4. $\models (x \in y) = (x = y)$ when $x, y \in \mathit{Var}$
5. $\models (x \in \neg\varphi) = \neg(x \in \varphi)$
6. $\models (x \in \varphi_1 \wedge \varphi_2) = (x \in \varphi_1) \wedge (x \in \varphi_2)$
7. $\models (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$, with x and y distinct
8. $\models x \in \sigma(\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1, ..., \varphi_{i-1}, y, \varphi_{i+1}, ..., \varphi_n))$

4.4 Defining special relations

Here we show how to define special relations using patterns. For example, $\exists y\,.\,\sigma(x_1, \ldots, x_n) = y$ states that $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a function in all models. Indeed, if M is any model satisfying the pattern above and $a_1 \in M_{s_1}$, $\ldots$, $a_n \in M_{s_n}$ then let $\rho : \mathit{Var} \to M$ be an M-valuation such that $\rho(x_1) = a_1, \ldots, \rho(x_n) = a_n$. Since M satisfies the pattern, it follows that $M_s = \bigcup\{\overline{\rho'}(\sigma(x_1, \ldots, x_n) = y) \mid \rho' : \mathit{Var} \to M,\ \rho'\restriction_{\mathit{Var}\setminus\{y\}} = \rho\restriction_{\mathit{Var}\setminus\{y\}}\}$. Since $\overline{\rho'}(\sigma(x_1, \ldots, x_n) = y)$ is either M_s or $\emptyset$, depending upon whether $\sigma_M(x_1, \ldots, x_n) = \{\rho'(y)\}$ holds or not, we conclude that there exists some $\rho' : \mathit{Var} \to M$ such that $\sigma_M(a_1, \ldots, a_n) = \{\rho'(y)\}$, that is, $\sigma_M(a_1, \ldots, a_n)$ is a one-element set. Therefore, σ_M represents a total function. To avoid writing such boring function patterns, from now on we automatically assume such an axiom whenever we write a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ using the function notation $\sigma : s_1 \times \cdots \times s_n \to s$.

Pattern $(f(x) = f(y)) \to (x = y)$ states that f is injective. If $(M, f_M : M \to M)$ is any model satisfying this specification, then f_M must be injective. Indeed, let $a, b \in M$ such that $a \neq b$ and $f_M(a) = f_M(b)$. Pick $\rho : \mathit{Var} \to M$ such that $\rho(x) = a$ and $\rho(y) = b$. Since M satisfies the axiom above, we get $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$. But Proposition 11 implies that $\overline{\rho}(x = y) = \emptyset$ and $\overline{\rho}(f(x) = f(y)) = M$, which is a contradiction. We can also show that any model whose f is injective satisfies the axiom. Let $(M, f_M : M \to M)$ be any model such that f_M is injective. It suffices to show $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$ for any $\rho : \mathit{Var} \to M$, which follows by Proposition 11: if $\rho(x) = \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = M$, and if $\rho(x) \neq \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = \emptyset$ because f_M is injective.

From here on in the rest of the paper we take the freedom to write $\varphi \neq \varphi'$ instead of $\neg(\varphi = \varphi')$. With this, another way to capture the injectivity of f is $(x \neq y) \to (f(x) \neq f(y))$.

Pattern $(\sigma(x_1, \ldots, x_n) = \bot_s) \vee \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ states that $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a partial function, and from now on we use the notation (note the "$\rightharpoonup$" symbol) $\sigma : s_1 \times \cdots \times s_n \rightharpoonup s$ to automatically assume a pattern like the above for σ. For example, a division partial function which is undefined in all models when the denominator is 0 can be specified as:

$$\_/\_ : \mathit{Nat} \times \mathit{Nat} \rightharpoonup \mathit{Nat} \qquad\qquad \neg(x/0)$$

i.e., as a symbol $\_/\_ \in \Sigma_{\mathit{Nat} \times \mathit{Nat}, \mathit{Nat}}$ with patterns $(x/y = \bot_{\mathit{Nat}}) \vee \exists z \,.\, x/y = z$ and $\neg(x/0)$.

Total relations can be defined with $\lceil \sigma(x_1, \ldots, x_n) \rceil_s^s$, equivalent to $\sigma(x_1, \ldots, x_n) \neq \bot_s$. We write $\sigma : s_1 \times \cdots \times s_n \Rightarrow s$ to automatically state that σ is a total relation.

4.5 Algebraic specifications and matching logic modulo theories

An algebraic specification is a many-sorted signature (S, Σ) together with a set of equations E over Σ-terms with variables. To translate an algebraic specification into a matching logic specification we only need to ensure that symbols get a function interpretation as described in Section 4.4, and to regard each equation $t = t'$ as an equality pattern $t = t'$.

▶ Proposition 13. Let (S, Σ, F) be the matching logic specification associated to the algebraic specification (S, Σ, E) as above. Then for any Σ-equation e, we have $E \models_{alg} e$ iff $F \models e$.

Using the notations introduced so far, Peano natural numbers can be defined as follows:

$$\begin{array}{lll} 0 : \to \mathit{Nat} & \mathit{succ} : \mathit{Nat} \to \mathit{Nat} & \mathit{plus} : \mathit{Nat} \times \mathit{Nat} \to \mathit{Nat} \\ \mathit{plus}(0, y) = y & & \mathit{plus}(\mathit{succ}(x), y) = \mathit{succ}(\mathit{plus}(x, y)) \end{array}$$

This looks identical to the conventional algebraic specification definition.

Note, however, that matching logic allows us to add more than just equational patterns. For example, we can add to F the pattern $0 \vee \exists x \,.\, \mathit{succ}(x)$ stating that any number is either 0 or the successor of another number. Nevertheless, since matching logic ultimately has the same expressive power as predicate logic (Proposition 7), we cannot finitely axiomatize in matching logic any mathematical domains that do not already admit finite FOL axiomatizations. In practice, we follow the same standard approach as the first-order SMT solvers, namely desired domains are theoretically presented with potentially infinitely many axioms but are implemented using specialized decision procedures. Indeed, our current matching logic implementation prototype in $\mathbb{K}$ defers to Z3 [5] the solving of all the domain constraints.

Algebraic specifications and decision procedures of mathematical domains abound in the literature. All of these can be used in the context of matching logic. We do not discuss these further, but only mention that from now on we tacitly assume definitions of integer and of natural numbers, as well as of Boolean values, with common operations on them. We assume that these come with three sorts, *Int*, *Nat* and *Bool*, and the operations on them use the conventional syntax and writing; e.g., $\_ \leq \_ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$, $x \leq y$, etc. To compact writing, we take the freedom to write b instead of $b = \mathit{true}$ for Boolean expressions b, in any sort context. For example, we write $\varphi_s \wedge x \leq y$ instead of $\varphi_s \wedge (x \leq y =^{s}_{\mathit{Bool}} \mathit{true})$.

4.6 Sequences, Multisets and Sets

Sequences, multisets and sets are typical ADTs. Matching logic enables, however, some useful developments and shortcuts. For simplicity, we only discuss collections over *Nat*, and name the corresponding sorts *Seq*, *MultiSet*, and *Set*. Ideally, we would build upon an order-sorted algebraic signature setting, so that we can regard $x{:}\mathit{Nat}$ not only as an element of sort *Nat*, but also as one of sort *Seq* (a one-element sequence), as one of sort *MultiSet*, as well as one of sort *Set*. Extending matching logic to an order-sorted setting is not difficult, but would deviate from our main objective in this paper, so we refrain from doing it. Instead, we rely on the reader to assume either that order-sortedness does not bring complications (besides those of order-sortedness itself in the context of algebraic specification) or that elements of sort *Nat* used in a *Seq*, *MultiSet*, or *Set* context are wrapped with injection symbols.

Sequences can be defined with two symbols and corresponding equations:

$$\epsilon : \to \mathit{Seq} \qquad \_\cdot\_ : \mathit{Seq} \times \mathit{Seq} \to \mathit{Seq} \qquad \epsilon \cdot x = x \qquad x \cdot \epsilon = x \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

We assume that lowercase variables have sort *Nat*, and uppercase ones have the appropriate collection sort. To avoid adding initiality constraints on models yet be able to do proofs by case analysis and elementwise equality, we may add $\epsilon \vee \exists x\,.\,\exists S.\, x \cdot S$ and $(x \cdot S = x' \cdot S') = (x = x') \wedge (S = S')$ as pattern axioms. We next define some operations on sequences:

$$\begin{array}{llll} \mathit{rev} : \mathit{Seq} \to \mathit{Seq} & \mathit{rev}(\epsilon) = \epsilon & \neg(x \in \epsilon) & x \in x \cdot S \\ \_ \in \_ : \mathit{Nat} \times \mathit{Seq} \to \mathit{Bool} & \mathit{rev}(x \cdot S) = \mathit{rev}(S) \cdot x & x \in y \cdot S \wedge (x \neq y) = x \in S & \end{array}$$

We can transform sequences into multisets adding the equality axiom $x \cdot y = y \cdot x$, and into sets by also including $x \cdot x = \bot$ or $x \cdot x = x$. Here is one way to axiomatize intersection:

$$\_\cap\_ : \mathit{Set} \times \mathit{Set} \to \mathit{Set} \qquad \epsilon \cap S_2 = \epsilon \qquad (x \cdot S_1) \cap S_2 = ((x \in S_2 \to x) \wedge (\neg(x \in S_2) \to \epsilon)) \cdot (S_1 \cap S_2)$$

4.7 Maps and Map Patterns

Finite-domain maps are also a typical example of an ADT. We only discuss maps from natural numbers to natural numbers, but they can be similarly defined over arbitrary domains as keys and as values. We use a syntax for maps that resembles that of separation logic [11]:

$$\begin{array}{ll} \_ \mapsto \_ : \mathit{Nat} \times \mathit{Nat} \rightharpoonup \mathit{Map} & \mathit{emp} * H = H \\ \mathit{emp} : \to \mathit{Map} & H_1 * H_2 = H_2 * H_1 \\ \_ * \_ : \mathit{Map} \times \mathit{Map} \rightharpoonup \mathit{Map} & (H_1 * H_2) * H_3 = H_1 * (H_2 * H_3) \\ 0 \mapsto a = \bot & x \mapsto a * x \mapsto b = \bot \end{array}$$

When regarding the above ADT as a matching logic specification, we can prove that the bottom two pattern equations above are equivalent to $\neg(0 \mapsto a)$ and, respectively, $(x \mapsto a * y \mapsto b) \to x \neq y$, giving the $\_ \mapsto \_$ and $\_ * \_$ the feel of "predicates".

Consider the canonical model of partial maps M, where: $M_{\mathit{Nat}} = \{0, 1, 2, \ldots\}$; $M_{\mathit{Map}} =$ partial maps from natural numbers to natural numbers with finite domains and undefined in

0, with emp interpreted as the map undefined everywhere, with $\_ \mapsto \_$ interpreted as the corresponding one-element partial map except when the first argument is 0 in which case it is undefined (note that $\_ \mapsto \_$ was declared using $\rightharpoonup$), and with $\_ * \_$ interpreted as map merge when the two maps have disjoint domains, or undefined otherwise (note that $\_ * \_$ was also declared using $\rightharpoonup$). M satisfies all axioms above.

We next define two common patterns, for complete linked lists and for list fragments:

$$\begin{array}{ll}
\mathit{list} : \mathit{Nat} \Rightarrow \mathit{Map} & \mathit{lseg} : \mathit{Nat} \times \mathit{Nat} \Rightarrow \mathit{Map} \\
\mathit{list}(0) = \mathit{emp} & \mathit{lseg}(x, x) = \mathit{emp} \\
\mathit{list}(x) \wedge x \neq 0 = \exists z \,.\, x \mapsto z * \mathit{list}(z) & \mathit{lseg}(x, y) \wedge x \neq y = \exists z \,.\, x \mapsto z * \mathit{lseg}(z, y)
\end{array}$$

It can be shown that in the model M of partial maps described above, there is a unique way to interpret *list* and *lseg*, namely as the expected linked lists and, respectively, linked list fragments. Specifically, we can show that $\mathit{lseg}_M : M_{\mathit{Nat}} \times M_{\mathit{Nat}} \to \mathcal{P}(M_{\mathit{Map}})$ (we only discuss *lseg*, because *list* is similar and simpler) can only be the following function:

$$\begin{array}{l}
\mathit{lseg}_M(n, n) = \{\mathit{emp}_M\} \text{ for all } n \geq 0 \\
\mathit{lseg}_M(n, m) = \{\ n \mapsto_M n_1 *_M n_1 \mapsto_M n_2 *_M \cdots *_M n_k \mapsto_M m \\
\qquad\qquad\qquad \mid k \geq 0, \text{ and } n_0 = n, n_1, n_2, \ldots, n_k > 0 \text{ all different}\}
\end{array}$$

Complete details can be found in [15].

It should be clear that patterns can be specified in many different ways. E.g., the first list pattern can also be specified as $\mathit{list}(x) = (x = 0 \wedge \mathit{emp} \vee \exists z \,.\, x \mapsto z * \mathit{list}(z))$. In a similar style, we can define more complex patterns, such as lists with data. But first, we specify a convenient operation for defining maps over contiguous keys, making use of a sequence data-type. The latter can be defined like in Section 4.6; for notational convenience, we take the freedom to use comma "," instead of "$\cdot$" for sequence concatenation in some places:

$$\_ \mapsto [\_] : \mathit{Nat} \times \mathit{Seq} \to \mathit{Map} \qquad x \mapsto [\epsilon] = \mathit{emp} \qquad x \mapsto [a, S] = x \mapsto a * (x + 1) \mapsto [S]$$

In our model M, we can take M_{Seq} to be the finite sequences of natural numbers, with ϵ_M and $\_ \cdot_M \_$ interpreted as the empty sequence and, respectively, sequence concatenation.

We can now define lists with data as follows:

$$\begin{array}{ll}
\mathit{list} : \mathit{Nat} \times \mathit{Seq} \Rightarrow \mathit{Map} & \mathit{lseg} : \mathit{Nat} \times \mathit{Seq} \times \mathit{Nat} \Rightarrow \mathit{Map} \\
\mathit{list}(0, \epsilon) = \mathit{emp} & \mathit{lseg}(x, \epsilon, x) = \mathit{emp} \\
\mathit{list}(x, n \cdot S) = \exists z \,.\, x \mapsto [n, z] * \mathit{list}(z, S) & \mathit{lseg}(x, n \cdot S, y) = \exists z \,.\, x \mapsto [n, z] * \mathit{lseg}(z, S, y)
\end{array}$$

Note that, unlike in the case of lists without data, this time we have not required the side conditions $x \neq 0$ and $x \neq y$, respectively. The side conditions were needed in the former case because without them we can infer, e.g., $\mathit{list}(0) = \bot$ (from the second equation of *list*), which using the first equation would imply $\mathit{emp} = \bot$. However, they are not needed in the latter case because it is safe (and even desired) to infer $\mathit{list}(0, n \cdot S) = \bot$ for any n and S. We can show, using a similar approach like for lists without data, that the pattern $\mathit{lseg}(x, S, y)$ matches in M precisely the lists starting with x, exiting to y, and holding data sequence S.

We can similarly define other data-type specifications, such as trees with data:

$$\begin{array}{lll}
\mathit{none} : \to \mathit{Tree} & \mathit{node} : \mathit{Nat} \times \mathit{Tree} \times \mathit{Tree} \to \mathit{Tree} & \mathit{tree} : \mathit{Nat} \times \mathit{Tree} \Rightarrow \mathit{Map} \\
\mathit{tree}(0, \mathit{none}) = \mathit{emp} & \multicolumn{2}{l}{\quad \mathit{tree}(x, \mathit{node}(n, t_1, t_2)) = \exists y\, z \,.\, x \mapsto [n, y, z] * \mathit{tree}(y, t_1) * \mathit{tree}(z, t_2))}
\end{array}$$

Therefore, fixing the interpretations of the basic mathematical domains, such as those of natural numbers, sequences, maps, etc., suffices in order to define interesting map patterns that appear in verification of heap properties of programs, in the sense that the axioms themselves uniquely define the desired data-types. No inductive predicates or principles were needed to define them, although induction or initiality may be needed in order to define the desired models. Choosing the right basic mathematical domains is, however, crucial. For example, if we allow the maps in M_{Map} to have infinite domains then the list patterns without data above (the first ones) also include infinite lists. The lists with data cannot include infinite lists, because we only allow finite sequences. This would, of course, change if we allow infinite sequences, or streams, in the model. In that case, *list* and *lseg* would not admit unique interpretations anymore, because we can interpret them to be either all the finite domain lists, or both the finite and the infinite-domain lists. Writing patterns which admit the desired solution in the desired model suffices in practice; our reasoning techniques developed in the sequel allow us to derive properties that hold in all models satisfying the axioms, so any derived property is sound also for the intended model and interpretations.

4.8 First-Order Logic

First-order logic (FOL) allows both function and predicate symbols:

$$\begin{array}{lcl} t_s & ::= & x \in Var_s \;\mid\; \sigma(t_1, \ldots, t_n) \;\text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \varphi & ::= & \pi(x_1, \ldots, x_n) \;\text{ with } \pi \in \Pi_{s_1 \ldots s_n} \;\mid\; \neg\varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \exists x.\varphi \end{array}$$

Let (S, Σ, Π) be a FOL signature. Like in pure predicate logic, we add a *Pred* sort and regard the predicate symbols as symbols of result *Pred*. Let (S^{ML}, Σ^{ML}) be the matching logic signature with $S^{ML} = S \cup \{Pred\}$ and $\Sigma^{ML} = \Sigma \cup \{\pi : s_1 \ldots s_n \to Pred \mid \pi \in \Pi_{s_1 \ldots s_n}\}$, and let F be $\{\exists z{:}s\,.\,\sigma(x_1{:}s_1, \ldots, x_n{:}s_n) = z \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$ saying each symbol is a function.

▶ Proposition 14. For any FOL formula φ, we have $\models_{FOL} \varphi$ iff $F \models \varphi$.

4.9 Separation Logic

Matching logic has inherent support for structural separation, without a need for any special logic constructs or extensions. That is because pattern matching has a spatial meaning by its very nature: matching a subterm already separates that subterm from the rest of the context, so matching two or more terms can only happen when there is no overlapping between them.

Separation logic [11, 14, 12] is a logic for reasoning about heap structures. There are many variants, but here we only consider the one in [11]. Its syntax extends FOL as follows:

$$\varphi \;::=\; \textit{(FOL syntax)} \;\mid\; emp \;\mid\; Int \mapsto Int \;\mid\; \varphi * \varphi \;\mid\; \varphi \mathrel{-\!\!*} \varphi$$

Its semantics is based on a fixed model of stores and heaps, which are finite-domain maps from variables and locations (particular integers), respectively, to integers. The semantics of each construct is given in terms of a pair (s, h) of a store and a heap, called a *state*. For example, $(s, h) \models_{SL} E_1 \mapsto E_2$ iff $Dom(h) = \overline{s}(E_1)$ and $h(\overline{s}(E_1)) = \overline{s}(E_2)$, and $(s, h) \models_{SL} P_1 * P_2$ iff there exist h_1 and h_2 of disjoint domains such that $h = h_1 * h_2$ and $(s, h_1) \models_{SL} P_1$ and $(s, h_2) \models_{SL} P_2$. The semantics of "magic wand", $P_1 \mathrel{-\!\!*} P_2$ is defined as the states whose heaps extended with a fragment satisfying P_1 result in ones satisfying P_2: $(s, h) \models_{SL} P_1 \mathrel{-\!\!*} P_2$ iff for any h_1 of domain disjoint of h's, if $(s, h_1) \models_{SL} P_1$ then $(s, h * h_1) \models_{SL} P_2$.

We can define a matching logic specification and a model of it, which precisely capture separation logic. The FOL constructs are already captured by the generic syntax of patterns as explained in previous sections. The spatial constructs, except for the $\mathrel{-\!\!*}$, are given by the matching logic specification of maps discussed in Section 4.7, in which we substitute *Int* for *Nat*. For the magic wand, we add the partial function $\_ \mathrel{-\!\!*} \_ : Map \times Map \rightharpoonup Map$ and the pattern $P_1 \mathrel{-\!\!*} P_2 = \exists H\,.\,H \wedge \lfloor H * P_1 \to P_2 \rfloor$. Recall from Section 4.3 that $\lfloor \_ \rfloor$ leverages the non-emptyness of its argument to the total set. In words, $P_1 \mathrel{-\!\!*} P_2$ is the set of all maps h

which merged with maps satisfying P_1 yield only maps satisfying P_2. With the above, any separation logic formula can be regarded, *as is*, as a matching logic pattern of sort *Map*.

We next construct our model. Let M be identical to the model for maps in Section 4.7, except that we replace natural numbers with integer numbers. The only thing left is to define the partial function $\_{-\!\!*}_M\_ : \mathit{Map} \times \mathit{Map} \to \mathcal{P}(\mathit{Map})$, which we do as follows: $h_1 {-\!\!*}_M\, h_2 = \{h \mid \mathit{Dom}(h) \cap \mathit{Dom}(h_1) = \emptyset$ and $h *_M h_1 = h_2\}$. Note that $h_1 {-\!\!*}_M\, h_2$ is either the empty set or it is a set of precisely one map. Then the following result holds:

▶ Proposition 15. If φ is a separation logic formula, then $\models_{SL} \varphi$ iff $M \models \varphi$. More specifically, for any store s and any heap h, we have $(s, h) \models_{SL} \varphi$ iff $h \in \overline{s}(\varphi)$.

5 Sound and Complete Deduction

As shown in Section 3, the proof system of predicate logic is sound for matching logic as is. Ideally, we would like the same to hold true for FOL with equality, that is, we would like its proof system to be sound as is for matching logic reasoning, where we replace terms and predicates with arbitrary patterns. Unfortunately, FOL's Substitution axiom, $(\forall x \,.\, \varphi) \to \varphi[t/x]$, is not sound if we replace t with any pattern. For example, consider the tautology $\forall x \,.\, \exists y \,.\, x = y$ and let φ be $\exists y \,.\, x = y$. If FOL's Substitution were sound for arbitrary patterns φ' instead of t, then the formula $\exists y \,.\, \varphi' = y$, stating that φ' evaluates to a unique element for any valuation, would be valid for any pattern φ'. However, this is not true in matching logic, because patterns can evaluate to any set of elements, including the empty set or the total set; several examples of such patterns were discussed in Section 4. We need to modify Substitution to indicate that φ' admits unique evaluations:

Substitution: $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$

Condition $\exists y \,.\, \varphi' = y$ holds when φ' is a term built with symbols σ obeying the functional axioms $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ discussed in Section 4.4. So the constrained substitution axiom is still more general than the original substitution axiom in FOL, since it can also apply when φ' is not built only from functional symbols but can be proved to have unique evaluation. It is interesting to note that a similar modification of Substitution was needed in the context of partial FOL [7], where the interpretations of functional symbols are partial functions, so terms may be undefined; axiom PFOL5 in [7] requires φ' to be *defined* in the Substitution rule, and several rules for proving definedness are provided. Note that our condition $\exists y \,.\, \varphi' = y$ is equivalent to definedness in the special case of PFOL, and that, thanks to the definability of equality in matching logic, we do not need special machinery for proving definedness.

Our approach to obtain a sound and complete proof system for matching logic is to build upon its reduction to predicate logic in Section 3. Specifically, to use Proposition 7 and the complete proof system of predicate logic. Given a matching logic signature (S, Σ), let (S, Π_Σ) be the predicate logic signature obtained like in Section 3. In addition to the *PL* translation there, we also define a backwards translation *ML* of (S, Π_Σ)-formulae into (S, Σ)-patterns:

$$\begin{array}{rcl}
\mathit{ML}(x = r) & = & x = r \\
\mathit{ML}(\pi_\sigma(r_1, \ldots, r_n, r)) & = & r \in \sigma(r_1, \ldots, r_n) \\
\mathit{ML}(\neg\psi) & = & \neg \mathit{ML}(\psi) \\
\mathit{ML}(\psi_1 \wedge \psi_2) & = & \mathit{ML}(\psi_1) \wedge \mathit{ML}(\psi_2) \\
\mathit{ML}(\exists x \,.\, \psi) & = & \exists x \,.\, \mathit{ML}(\psi) \\
\mathit{ML}(\{\psi_1, \ldots, \psi_n\}) & = & \{\mathit{ML}(\psi_1), \ldots, \mathit{ML}(\psi_n)\}
\end{array}$$

Recall from Section 4.3 that we assume equality and membership in all specifications.

FOL axioms and rules:

1. $\vdash$ propositional tautologies
2. Modus ponens: $\vdash \varphi_1$ and $\vdash \varphi_1 \to \varphi_2$ imply $\vdash \varphi_2$
3. $\vdash (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ when $x \notin FV(\varphi_1)$
4. Universal generalization: $\vdash \varphi$ implies $\vdash \forall x \,.\, \varphi$
5. Substitution: $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$
6. Equality introduction: $\vdash \varphi = \varphi$
7. Equality elimination: $\vdash \varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$

Membership axioms and rules:

8. $\vdash \forall x \,.\, x \in \varphi$ iff $\vdash \varphi$
9. $\vdash x \in y = (x = y)$ when $x, y \in \mathit{Var}$
10. $\vdash x \in \neg\varphi = \neg(x \in \varphi)$
11. $\vdash x \in \varphi_1 \wedge \varphi_2 = (x \in \varphi_1) \wedge (x \in \varphi_2)$
12. $\vdash (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$, with x and y distinct
13. $\vdash x \in \sigma(\varphi_1,.., \varphi_{i-1}, \varphi_i, \varphi_{i+1},.., \varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1,.., \varphi_{i-1}, y, \varphi_{i+1},.., \varphi_n))$

Figure 2 Sound and complete proof system of matching logic.

Figure 2 shows our sound and complete proof system for matching logic reasoning, which was specifically crafted to include the proof system of first-order logic. Indeed, the first group of axiom and rule schemas include all the axioms and proof rules of FOL with equality as instances (the rules Substitution, Equation introduction and Equation elimination allow more general patterns instead of terms). The second group of proof rules, for reasoning about membership, is introduced for technical reasons, namely for the proof of Theorem 16:

▶ **Theorem 16.** *The proof system in Figure 2 is sound and complete: $F \models \varphi$ iff $F \vdash \varphi$.*

6 Additional Related Work

Matching logic builds upon intuitions from and relates to at least four important logical frameworks: (1) *Relation algebra (RA)* (see, e.g., [21]), noticing that our interpretations of symbols as functions to powersets are equivalent to relations; although our interpretation of symbols captures better the intended meaning of pattern and matching, and our proof system is quite different from that of RA, like with FOL we expect a tight relationship between matching logic and RA, which is left as future work; (2) *Partial FOL* (see, e.g., [7] for a recent work and a survey), noticing that our interpretations of symbols into powersets are more general than partial functions (Section 4.3 shows how we defined definedness); and (3) *Separation logics (SL)* (see, e.g.,[11]), which we briefly discussed in Section 4.9 but refer the reader to [15] for more details; and (4) Precursors of matching logic in [17, 20, 16], which proposed the pattern idea by extending FOL with particular "configuration" terms:

$$\begin{array}{rcl} t_s & ::= & x \in \mathit{Var}_s \quad | \quad \sigma(t_1, \ldots, t_n) \ \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \varphi & ::= & \pi(x_1, \ldots, x_n) \ \text{ with } \pi \in \Pi_{s_1 \ldots s_n} \quad | \quad \neg\varphi \quad | \quad \varphi \wedge \varphi \quad | \quad \exists x.\varphi \\ & | & t \in T_{\Sigma, \mathit{Cfg}}(X) \end{array}$$

where $T_{\Sigma,\mathit{Cfg}}(X)$ is the set of terms of a special sort *Cfg* (from "configurations") over variables in set X. To avoid naming conflicts, we propose to call the variant above *topmost matching* *logic* from here on. Topmost matching logic can trivially be desugared into FOL with equality by regarding a particular pattern predicate $t \in T_{\Sigma,\mathit{Cfg}}(X)$ as syntactic sugar for "(current state/configuration is) equal to t". One major limitation of topmost matching logic, which motivated the generalization in this paper, is that its restriction to patterns of sort *Cfg* prevented us to define local patterns (e.g., the heap list pattern) and perform local reasoning.

The idea of regarding arbitrary terms as patterns is reminiscent to *pattern calculus* [10], although note that matching logic's patterns are intended to express and reason about static properties of data-structures or program configurations, while pattern calculi are aimed at generally and compactly expressing computations and dynamic behaviors of systems. So far we used rewriting to define dynamic language semantics; it would be interesting to explore the combination of pattern calculus and matching logic for language semantics and reasoning.

7 Conclusion and Future Work

Matching logic is a sound and complete FOL variant that makes no distinction between function and predicate symbols. Its formulae, called patterns, mix symbols, logical connectives and quantifiers, and evaluate in models to sets of values, those that "match" them, instead of just one value as terms do or a truth value as predicates do in FOL. Equality can be defined and several important variants of FOL fall as special fragments. Separation logic can be framed as a matching logic theory within the particular model of partial finite-domain maps, and heap patterns can be elegantly specified using equations. Matching logic allows spatial specification and reasoning anywhere in a program configuration, and for any language, not only in the heap or other particular and fixed semantic components.

We made no efforts to minimize the number of rules in our proof system, because our main objective here was to include the proof system for FOL with equality. It is likely that a minimal proof system working directly with the core symbols $\lceil\_\rceil_{s_1}^{s_2} \in \Sigma_{s_1,s_2}$ for all sorts $s_1, s_2 \in S$ can be obtained such that the equality and membership axioms and rules in Figure 2 can be proved as lemmas. Likewise, we refrained from discussing any computationally effective fragments of matching logic, although we are implementing them in $\mathbb{K}$. Finally, complexity results in the style of [1, 3, 9] for separation logic can likely also be obtained for fragments of matching logic.

Acknowledgments. I wish to express my deepest thanks to the $\mathbb{K}$ team (`http://kframework.org`), who share the belief that programming languages should have only one semantics, which should be executable, and formal analysis tools, including fully fledged deductive program verifiers, should be obtained from such semantics at little or no extra cost. I would like to also warmly thank the following colleagues and friends for their comments and criticisms on previous drafts of this paper: Nikolaj Bjorner, Claudia-Elena Chiriţă, Maribel Fernández, Dorel Lucanu, Brandon Moore, Cosmin Rădoi, Andrei Ştefănescu.

References

1 Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *FOSSACS'14*, LNCS 8412, pages 411–425, 2014.

2 Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

3 James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. Technical Report RN/13/15, University College London, 2013.
4 Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.
5 Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008. LNCS 4963.
6 Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
7 William M. Farmer and Joshua D. Guttman. A set theory with support for partial functions. *Studia Logica*, 66(1):59–78, 2000.
8 Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 2015.
9 Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In *CADE'13*, LNCS 7898, 2013.
10 C. Barry Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, November 2004.
11 Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19. LNCS 2142, 2001.
12 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symb. Logic*, 5(2):215–244, 1999.
13 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 2015.
14 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
15 Grigore Roşu. Matching logic: A logic for structural reasoning. Technical Report http://hdl.handle.net/2142/47004, University of Illinois, Jan 2014.
16 Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS'13*. IEEE, 2013.
17 Grigore Rosu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010.
18 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
19 Grigore Rosu and Traian Florin Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.
20 Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 555–574. ACM, 2012.
21 A. Tarski and S.R. Givant. *A Formalization of Set Theory Without Variables.* Number 41. AMS, 1987.